



SMART CONTRACT AUDIT REPORT

for

SHOYU PROTOCOL



Prepared By: Yiqun Chen

PeckShield
July 26, 2021

Document Properties

Client	Shoyu
Title	Smart Contract Audit Report
Target	Shoyu
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 26, 2021	Xuxian Jiang	Final Release
1.0-rc1	July 25, 2021	Xuxian Jiang	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1	Introduction	4
1.1	About Shoyu	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	7
2	Findings	9
2.1	Summary	9
2.2	Key Findings	10
3	Detailed Results	11
3.1	Repeated Execution Of Passed Proposals	11
3.2	Expiration Consistency On Proposal Execution	12
3.3	Improved Validation On Proposal Submission	13
3.4	Proper Signature Validation in BaseExchange	15
3.5	Trust Issue of Admin Keys	16
3.6	Outdated bestBid Removal	17
3.7	Exchange Bypass With Direct ERC721 safeTransferFrom()	18
4	Conclusion	20
	References	21

1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the Shoyu protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Shoyu

Shoyu is the Sushi NFT Exchange and Launchpad, which is an innovative NFT platform that focuses on artists and allows creators to push the space forward. Specifically, current NFTs are limited by file format and sizing for artists and their creative abilities. And the Shoyu platform is designed to be beautiful, functional and have the Japanese sensibility and aesthetic. The platform has an ambitious goal of allowing artists and creators to create NFTs that they are unable to make on other platform and enabling collectors to curate and showcase their digital arts.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Shoyu

Item	Description
Target	Shoyu
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 26, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- <https://github.com/sushiswap/shoyu.git> (d5b1aac)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- <https://github.com/sushiswap/shoyu.git> (0336010)

1.2 About PeckShield

PeckShield Inc. [11] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (<https://t.me/peckshield>), Twitter (<http://twitter.com/peckshield>), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [9], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.




Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `Shoyu` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	
Medium	3	
Low	3	
Informational	0	
Total	7	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in [Section 3](#).

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 3 medium-severity vulnerabilities, and 3 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Repeated Execution Of Passed Proposals	Business Logic	Fixed
PVE-002	Medium	Expiration Consistency On Proposal Execution	Business Logic	Fixed
PVE-003	High	Improved Validation On Proposal Submission	Business Logic	Fixed
PVE-004	Low	Proper Signature Validation in BaseExchange	Coding Practices	Fixed
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-006	Low	Outdated bestBid Removal	Coding Practices	Fixed
PVE-007	Low	Exchange Bypass With Direct ERC721 safeTransferFrom()	Business Logic	Fixed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Repeated Execution Of Passed Proposals

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Low
- Target: ERC721GovernanceToken
- Category: Business Logic [8]
- CWE subcategory: CWE-837 [4]

Description

The Shoyu protocol supports the fractional ownership that allows users who have been previously priced out of certain NFTs or artists (such as Bleeple) to be able to buy a piece of their work. And token holders can vote for a sell proposal to run the NFT governance. Our analysis on the proposal-execution logic shows a proposal, if successfully passed, may allow for repeated execution.

To elaborate, we show below the `executeSellProposal()` function. It implements a rather straightforward logic in validating that this proposal is indeed successfully passed and then executing the proposal with the helper of an internal routine `_executeSellProposal()`. It comes to our attention that an executed proposal may be executed multiple times!

```

168     function executeSellProposal(uint256 id) external override {
169         SellProposal storage proposal = proposals[id];
170         require(proposal.expiration < block.number, "SHOYU: NOT_FINISHED");
171         require(totalPowerOf[id] > _minPower(), "SHOYU: NOT_SUBMITTED");
172
173         _executeSellProposal(proposal);
174
175         emit ExecuteSellProposal(id);
176     }
177
178     function _minPower() internal view returns (uint256) {
179         return (TOTAL_SUPPLY * minimumQuorum) / 100;
180     }
181
182     function _executeSellProposal(SellProposal storage proposal) internal {

```

```

183     proposal.executed = true;
184
185     IOrderBook(orderBook).submitOrder(
186         nft,
187         tokenId,
188         1,
189         proposal.strategy,
190         proposal.currency,
191         address(0),
192         proposal.deadline,
193         proposal.params
194     );
195 }

```

Listing 3.1: ERC721GovernanceToken::executeSellProposal()

Recommendation Revise the above `executeSellProposal()` function to ensure the proposal, if passed, can be executed only once.

Status The issue has been fixed by this commit: [55e2b7c](#).

3.2 Expiration Consistency On Proposal Execution

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Low
- Target: ERC721GovernanceToken
- Category: Business Logic [8]
- CWE subcategory: CWE-837 [4]

Description

As mentioned in Section 3.1, the Shoyu protocol supports the fractional ownership of NFTs and allows token holders to vote for a sell proposal to run the NFT governance. While analyzing the proposal lifecycle, we notice there is an inconsistency regarding the proposal expiration.

In particular, if we examine the following two functions `executeSellProposal()` and `confirmSellProposal()`. The first function is designed to execute a successfully-passed proposal and the second function is used to vote on a proposal. According to the first function, the proposal can be executed only when it is expired. However, according to the second function, the proposal can be executed before it is expired as long as the vote has more than the `minimumQuorum` balance power.

```

168     function executeSellProposal(uint256 id) external override {
169         SellProposal storage proposal = proposals[id];
170         require(proposal.expiration < block.number, "SHOYU: NOT_FINISHED");
171         require(totalPowerOf[id] > _minPower(), "SHOYU: NOT_SUBMITTED");
172     }

```

```

173     _executeSellProposal(proposal);
174
175     emit ExecuteSellProposal(id);
176 }

```

Listing 3.2: ERC721GovernanceToken::executeSellProposal()

```

132     function confirmSellProposal(uint256 id) external override {
133         SellProposal storage proposal = proposals[id];
134         require(!proposal.executed, "SHOYU: EXECUTED");
135         require(block.number <= proposal.expiration, "SHOYU: EXPIRED");
136         require(totalPowerOf[id] > 0, "SHOYU: NOT_SUBMITTED");
137         require(powerOf[id][msg.sender] == 0, "SHOYU: CONFIRMED");
138
139         uint256 power = balanceOfAt(msg.sender, proposal.snapshotId);
140         require(power > 0, "SHOYU: INSUFFICIENT_POWER");
141
142         totalPowerOf[id] += power;
143         powerOf[id][msg.sender] = power;
144
145         emit ConfirmSellProposal(id, msg.sender, power);
146
147         if (totalPowerOf[id] > _minPower()) {
148             _executeSellProposal(proposal);
149
150             emit ExecuteSellProposal(id);
151         }
152     }

```

Listing 3.3: ERC721GovernanceToken::confirmSellProposal()

Recommendation Be consistent on the conditions when the proposal is ready for execution.

Status The issue has been fixed by this commit: [f00988f](#).

3.3 Improved Validation On Proposal Submission

- ID: PVE-003
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: ERC721GovernanceToken
- Category: Business Logic [8]
- CWE subcategory: CWE-837 [4]
- CWE subcategory: CWE-841 [5]

Description

As mentioned earlier, the shoyu protocol supports the fractional ownership of NFTs and allows token holders to vote for a sell proposal to run the NFT governance. The previous sections focus on the

proposal execution logic. In this section, we examine the proposal submission logic and report a possible flashloan-assisted issue to bypass the `minimumQuorum` restriction. Note that if the proposal under submission has accumulated more than `minimumQuorum` balance power, the protocol may kick off the selling process.

To elaborate, we show below the `submitSellProposal()` function. This function implements a rather straightforward logic in recording the new proposal submission information and computing the current power with the proposer's balance. It comes to our attention that the proposer may make a flashloan to dramatically increase the balance and hence increase the voting power beyond the required `minimumQuorum` restriction. In other words, the proposal can be trivially passed.

```

110     function submitSellProposal(
111         address strategy,
112         address currency,
113         uint256 deadline,
114         bytes calldata params,
115         uint256 expiration
116     ) external override {
117         require(block.number < expiration, "SHOYU: EXPIRED");
118
119         uint256 power = balanceOf(msg.sender);
120         require(power > 0, "SHOYU: INSUFFICIENT_POWER");
121
122         uint256 id = proposals.length;
123         uint256 snapshotId = _snapshot();
124
125         proposals.push(SellProposal(false, strategy, currency, deadline, params,
126             expiration, snapshotId));
127         totalPowerOf[id] = power;
128         powerOf[id][msg.sender] = power;
129
130         emit SubmitSellProposal(id, snapshotId, msg.sender, power);
131     }

```

Listing 3.4: `ERC721GovernanceToken::submitSellProposal()`

Recommendation Develop necessary counter-measures (e.g., in limiting the proposer to be an EOA-based account) to block the above-mentioned flashloan attack.

Status The issue has been fixed by this commit: [64afde1](#).

3.4 Proper Signature Validation in BaseExchange

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: BaseExchange
- Category: Coding Practices [7]
- CWE subcategory: CWE-563 [3]

Description

The Shoyu protocol has a built-in BaseExchange contract that, as the name indicated, has designed with the basic exchange functionality. In fact, Shoyu's ERC-721 and ERC-1155 contracts are equipped with the NFT exchange functionality in themselves. This support allows for convenient and gas-optimized exchange and gives the owner the full power to change configurations needed for trades.

The exchange functionality requires a proper way for the buyer and seller to confirm their identity. And this task is delegated to an internal helper routine `_verify()`. To elaborate, we show below this helper routine. This routine ensures that the given signer is indeed the one who signs the message. Note that if the `signer` is a contract, it will call back the `signer` to validate it. Otherwise, the `ecrecover()` precompile is used for validation. It comes to our attention that the precompile-based validation needs to properly ensure the `signer` is not equal to `address(0)`.

```

180     function _verify(
181         bytes32 hash,
182         address signer,
183         uint8 v,
184         bytes32 r,
185         bytes32 s
186     ) internal view {
187         bytes32 digest = keccak256(abi.encodePacked("\x19\x01", DOMAIN_SEPARATOR(), hash
188             ));
189         if (Address.isContract(signer)) {
190             require(
191                 IERC1271(signer).isValidSignature(digest, abi.encodePacked(r, s, v)) ==
192                 0x1626ba7e,
193                 "SHOYU: UNAUTHORIZED"
194             );
195         } else {
196             require(ecrecover(digest, v, r, s) == signer, "SHOYU: UNAUTHORIZED");
197         }
198     }

```

Listing 3.5: BaseExchange::_verify()

Recommendation Strengthen the `_verify()` routine to ensure the `signer` is not equal to `address(0)`.

Status The issue has been fixed by this commit: dba56b2.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [6]
- CWE subcategory: CWE-287 [2]

Description

In the Shoyu protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and marketing adjustment). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```

97     function setOperationalFee(uint8 operationalFee) external override onlyOwner {
98         require(operationalFee <= MAX_OPERATIONAL_FEE, "SHOYU: INVALID_FEE");
99
100         _operationalFee = operationalFee;
101     }
102
103     function setStrategyWhitelisted(address ask, bool whitelisted) external override
104         onlyOwner {
105         require(ask != address(0), "SHOYU: INVALID_SALE");
106
107         isStrategyWhitelisted[ask] = whitelisted;
108     }

```

Listing 3.6: TokenFactory::setOperationalFee()/setStrategyWhitelisted()

If the privileged `owner` account is a plain EOA account, this may be worrisome and pose counterparty risk to the exchange users. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation.

Moreover, it should be noted that current contracts have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks.

Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed with the team. For the time being, the team has confirmed that these privileged functions should be called by a trusted multi-sig account, not a plain EOA account.

3.6 Outdated bestBid Removal

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: BaseExchange
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

As mentioned in Section 3.4, the Shoyu protocol has a built-in BaseExchange contract and every single ERC-721/ERC-1155 token in Shoyu is automatically equipped with the NFT exchange functionality in itself. While examining the bidding logic, we notice the final NFT-claiming logic needs to be improved.

To elaborate, we show blow the related `claim()` function. While it properly transfers the NFT on sale to the best bidder, the current implementation does not clean up the `bestBid` information. Note that the left-behind `bestBid` information is considered outdated and should not be possible for later reuse. To eliminate the risk of being reused, there is a need to remove the outdated `bestBid` information.

```

144     function claim(Orders.Ask memory askOrder) external override nonReentrant {
145         require(canTrade(askOrder.token), "SHOYU: INVALID_EXCHANGE");
146         require(askOrder.deadline < block.number, "SHOYU: NOT_CLAIMABLE");
147
148         bytes32 askHash = askOrder.hash();
149         _validate(askOrder, askHash);
150         _verify(askHash, askOrder.signer, askOrder.v, askOrder.r, askOrder.s);
151
152         BestBid memory best = bestBid[askHash];
153         require(msg.sender == best.bidder, "SHOYU: FORBIDDEN");
154         require(ISTrategy(askOrder.strategy).canExecute(askOrder.params, best.price), "
            SHOYU: FAILURE");
155
156         amountFilled[askHash] += best.amount;
157
158         address bidRecipient = best.recipient;
159         if (bidRecipient == address(0)) bidRecipient = best.bidder;
160         _transfer(askOrder.token, askOrder.signer, bidRecipient, askOrder.tokenId, best.
            amount);

```

```

161
162     address recipient = askOrder.recipient;
163     if (recipient == address(0)) recipient = askOrder.signer;
164     _transferFeesAndFunds(askOrder.currency, best.bidder, recipient, best.price *
        best.amount);
165
166     emit Execute(askHash, best.bidder, best.amount, best.price, bidRecipient, best.
        referrer);
167 }

```

Listing 3.7: BaseExchange::claim()

Recommendation Revise the above claim() by removing the bestBid information when the related NFT is transferred to the best bidder.

Status The issue has been fixed by this commit: [aac1756](#).

3.7 Exchange Bypass With Direct ERC721 safeTransferFrom()

- ID: PVE-007
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: BaseExchange
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

Each NFT supported in the built-in BaseExchange is represented as an ERC721-based NFT token, which naturally has the standard implementation, e.g., transferFrom()/safeTransferFrom(). To optimize gas cost when submitting sell order, orders are mainly kept off-chain. When an account wants to buy the NFT, the user can pick the order with the maker's signature and execute it with a proper bid parameters such as the price. Note the current implementation supports NFT royalties that give the owner a percentage of the sale price each time their NFT creation is sold on Shoyu.

```

144     function claim(Orders.Ask memory askOrder) external override nonReentrant {
145         require(canTrade(askOrder.token), "SHOYU: INVALID_EXCHANGE");
146         require(askOrder.deadline < block.number, "SHOYU: NOT_CLAIMABLE");
147
148         bytes32 askHash = askOrder.hash();
149         _validate(askOrder, askHash);
150         _verify(askHash, askOrder.signer, askOrder.v, askOrder.r, askOrder.s);
151
152         BestBid memory best = bestBid[askHash];
153         require(msg.sender == best.bidder, "SHOYU: FORBIDDEN");
154         require(ISTrategy(askOrder.strategy).canExecute(askOrder.params, best.price), "
            SHOYU: FAILURE");

```

```

156         amountFilled[askHash] += best.amount;

158         address bidRecipient = best.recipient;
159         if (bidRecipient == address(0)) bidRecipient = best.bidder;
160         _transfer(askOrder.token, askOrder.signer, bidRecipient, askOrder.tokenId, best.
            amount);

162         address recipient = askOrder.recipient;
163         if (recipient == address(0)) recipient = askOrder.signer;
164         _transferFeesAndFunds(askOrder.currency, best.bidder, recipient, best.price *
            best.amount);

166         emit Execute(askHash, best.bidder, best.amount, best.price, bidRecipient, best.
            referrer);
167     }

```

Listing 3.8: BaseExchange::claim()

To elaborate, we show above the `claim()` routine. This routine is used for claiming a listed NFT with proper fee/royalty payment. It comes to our attention that instead of paying the fee/royalty amount, it is possible for the current owner and the buyer to directly negotiate a price, without paying the `protocolFeeRecipient` and `royaltyFeeRecipient`. The NFT can then be arranged and delivered by the current owner to directly call `transferFrom()/safeTransferFrom()` with the buyer as the recipient.

Recommendation Implement a locking mechanism so that any NFT, needs to be locked in the `BaseExchange` contract in order to be available for public auction.

Status The issue has been fixed by this commit: [cd3d49a](#).

4 | Conclusion

In this audit, we have analyzed the `Shoyu` design and implementation. The system presents a unique offering as a decentralized `Sushi NFT exchange` and `launchpad`. The current code base is clearly organized and those identified issues are promptly confirmed and resolved.

Meanwhile, we need to emphasize that `Solidity`-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.



References

- [1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. <https://cwe.mitre.org/data/definitions/1126.html>.
- [2] MITRE. CWE-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [3] MITRE. CWE-563: Assignment to Variable without Use. <https://cwe.mitre.org/data/definitions/563.html>.
- [4] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. <https://cwe.mitre.org/data/definitions/837.html>.
- [5] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. <https://cwe.mitre.org/data/definitions/841.html>.
- [6] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
- [7] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [8] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.
- [9] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.

- [10] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.
- [11] PeckShield. PeckShield Inc. <https://www.peckshield.com>.

